Graph visualization with OpenGL

Hannes Ahlgren
Abstract

*Vizz3D* is a 3D graphics code analysis tool, developed at Växjö University that optionally can use Java3D or OpenGL. However, initially Java3D was the only programming interface used. No other version was considered. Therefore the applications structure was built with the Java3D way of thought in mind. But code visualization with 3D graphics can be a demanding task for the computers processor and its graphics hardware and Java3D is known to be somewhat inefficient. So an OpenGL version was introduced.

This thesis reflects on the work restructuring the application’s code to fit both versions within *Vizz3D* in a structured and object-oriented way. The thesis shows the efforts to be taken to make an existing ever evolving tool easily extendible to other API’s. Additional aspects of OpenGL specific implementations are discussed throughout the thesis.
1 Introduction

Understanding of software code and its structures is an essential part of software engineering. One way to approach this is analysis and visualization. A common way to visualize code structures is UML (Unified Modelling Language) where class diagrams are a popular choice. The drawback with UML diagrams is that they easily get huge proportions and are difficult to follow even for a smaller visualized system.

Another approach is to build up a graph tree structure with nodes and edges that describe the structure. It can describe the folder system in which the classes are stored or how they actually work, where the nodes embody the classes and the edges are the nodes relations (e.g. a call or inheritance).

To enhance the understanding, 3D graphics with interactive capabilities is a good way to describe software. Ability to rotate the graph helps the user to understand and intuitively “read” the structure and its relations. Hiding certain sections or certain types helps the clarification of larger structures are connected together.

Various ways of visualizing the nodes and edges is a further aid. A package node could be embodied by a skyscraper while a class node could be a smaller building. The edges can take the form of roads or cables connecting the buildings. This would give the user a intuitive hierarchical environment familiar to the user.

1.1 Context

VizzAnalyzer [1] is a framework for reverse engineering, developed to analyse source code and visualizing it with different tools. The VizzAnalyzer uses a dynamic type system to control the data exchange between the tools, and a set of wrapper classes handle their communication between them. The idea is to allow different tools, existing and future ones, to integrate and work together. Some of the tools that have been adapted to the framework today are yED [13], a 2D graph editor, Wilma [14] a 3D graph tools, CrocoPat [15] which is a tool for relational computation and Recoder [16] which is a Java framework for source code metaprogramming.

Another plug-in is Vizz3D, a tool to present the analysed code in 3D graphics with the possibility to interact with the visualization. At first, all of the 3D graphics in Vizz3D was programmed using Java3D [7]. The latest work in this area is discussed in Rodrigo Cano Martins thesis [2]. The spring semester of 2004 an OpenGL [3] test version was created as a 3D graphics course project. The test version did not fully implement Vizz3D’s features but made a ground structure to work with.

This thesis covers the further development of the OpenGL version of Vizz3D visualization and interaction. It will be referred to as VizzGL throughout the thesis while the Java3D version will be referred to as VizzJ3d.

1.2 Motivation

Java3D, a 3D graphics API, is a good way of creating 3D graphics, since it is fairly easy to learn and most of all easy to apply when dealing with Java code, but it has some drawbacks. These drawbacks mainly are performance and lack of low level control.

Using native OpenGL is faster than Java3D; in fact Java3D uses OpenGL itself because of its good performance. When working with OpenGL one more or less has to define everything “by hand” which gives tremendous control but of course makes it more complex.
All in all, an OpenGL version was needed. Further explanations on Java3D and OpenGL can be found in chapter 2.

1.3 Problem
As there already exists an application that do all the visualization, the idea was to create an OpenGL version that could use some of the previous work, for saving time and to present a well structured object oriented code. But the initial implementation was not done with other graphics API’s in mind as it was never intended to be extendible for other API’s. This turned out to be the actual problem of this thesis: Vizz3D is not particularly extendible in respect to additional 3D API’s.

Therefore the goal of this thesis was identified as:

Refactor Vizz3D to make multiple 3D programming interfaces possible.

1.4 Goals and Criteria
The main goal for the thesis is to create an OpenGL version of Vizz3D that works as the Java3D one. Sub-goals are as follows:

1. Refactoring and Integration - The existing version of Vizz3D have a lot of methods and algorithms that just needs translation/refactoring. The aim was to as far as possible use this existing code. This also includes dividing the OpenGL code into several smaller files in an object oriented way.

2. Assembling – The second goal was to build a resembling scenegraph structure as in VizzJ3d. This is not the natural way to display things in OpenGL but on the other hand, when dealing with hundreds of objects there has to be some sort of graph structure and of course, this is the way it is displayed on screen.

3. Adaptation - VizzJ3d uses bindings to visualize the nodes and edges in different colors and metaphors. This had to be done in VizzGL as well. A lot of this work is predefined in Java3D but not in OpenGL so there had to be some new classes that described the shape, appearance and material.

4. Edge arrows - The edges have a direction. One of the goals was to find out a way to visualize this.

5. Interaction - Translations in the scene should have a “natural feel”. The goal was to create the translations react to the mouse dragging like one intuitively wants it to react.

Many of these sub-goals have their own sub-goals but it will be discussed in more detail in the implementation chapter.

1.5 Outline
Chapter 2 discusses the choices of 3D graphics API’s and a little of how they work. Chapter 3 describes the system architecture. Vizz3D has a complex structure which will be explained in this chapter. Chapter 4 discusses the implementation and the thoughts on how and why things were done in a specific way. Chapter 5 is a summary of the thesis and comments about future work.
2 3D Graphics API’s

This chapter will try to explain briefly what OpenGL is and how it works. Also a look at the GL4Java [5] wrapper will be presented since this is used when programming Vizz3D. A shorter presentation on how Java3D works is also included. The information is gathered from the OpenGL homepage [3] and The OpenGL SuperBible [4]

2.1 OpenGL

OpenGL is developed by SGI, Silicon Graphics Inc. 1992, and can be defined as a software interface to graphical hardware - or, if you wish, a fast and portable interface for computer graphics and modelling. It is not a programming language like C++ or Java. Instead one includes the OpenGL API (Application Program Interface) in the ordinary code in order to use its predefined functions, such as transformations, geometric primitives and so on. This is to simplify for the programmer. Also, all the calculations used in OpenGL are optimized to the limit.

2.1.1 Generic/Software Implementation

OpenGL can be implemented in two various ways; generic (software) or through hardware. The software implementation is an application that can be run practically anywhere as long as the system can handle the graphically generated image, for instance through Windows GDI (Graphics Device Interface). See figure 2.1.

An ordinary application program can use the programmers own functions or predefined functions from the OS (Operating System) or the programming language’s runtime library.

Figure 2.1 OpenGL’s place in a typical software implementation

2.1.2 Hardware Implementation

A typical example of a hardware implementation is the graphics card driver. The difference to the software implementation is that here, all calls to OpenGL goes via the hardware driver directly to the display, without involving the GDI (fig 2.2). Since a hardware implementation much faster than a software implementation it usually is called an accelerated implementation.
OpenGL have become a standard in computer graphics which have made the most, if not all, graphics cards adjust to OpenGL. Also, today’s graphics cards handle more and more of the calculations. This takes a big load off from the CPU and speeds up the processing.

2.1.3 The OpenGL Pipeline
Figure 2.3 describes the various steps in the OpenGL pipeline. In other words, what happens at OpenGL calls. First of all, the call is placed in a command buffer, waiting for its turn to be dealt with, flushed. If any transformation or lighting calculations are needed this is done. After this, a rasterization is done. This is more or less giving color to the image. Finally the image comes to the frame buffer which is the memory for the display device – the picture is on your screen.

2.1.4 The OpenGL State Machine
The state machine is a way to decide how things should be displayed. This is done by setting flags like “is the fog on” or “should the insides of a box be calculated” and similar states. There are numerous states and they all sum up to a rendering state which is read by the OpenGL functions.

2.1.5 GLU/GLUT
As mentioned before OpenGL’s creators strived to make it completely platform independent and therefore OpenGL lacks keyboard and window handling and some complex curves/surfaces.
calculations. This is however included in two extra libraries; the GLU (OpenGL Utility library) that handles common but complex tasks like predefined spheres, and then it is the GLUT (OpenGL Utility Toolkit) that mainly takes care of window handling and interaction. Both of these libraries have many more functions that makes coding OpenGL easier but it also makes the code platform dependant.

2.2 GL4Java

Since OpenGL is written in C/C++ there is no natural way to use OpenGL with Java code. Therefore several wrappers have been developed and one of them, the one used in this project, is Jausoft’s [5] GL4Java (OpenGL for Java). This is a wrapper that adds native OpenGL to the Java Virtual Machine (JVM). It is installed on the local hard disc and runs as a stand alone application. According to Jausoft, GL4Java should be as close to native OpenGL as any Java code can get. The performance depends on the underlying JVM and OpenGL implementations (software or hardware implementation).

GL4Java has two layers; the Java Classes and the Native Library. The integration of GL4Java is done with the Java Native Interface (JNI) and all OpenGL calls from Java go through JNI to the native OpenGL library (fig 2.4).

2.3 Java3D

2.3.1 Background

This thesis will not go deeply into the Java3D API but it shall be explained roughly how it works in a large scale since the work is more or less a translation/refactoring to OpenGL from a Java3D
application of Vizz3D. The information is collected from Rüdiger Lincke’s master thesis [6] and the Java3D homepage [7].

Java3D is developed at Sun Microsystems and the main goal was to create a high level 3D graphics API for Java. It uses OpenGL or DirectX to do the low level calculations, the native rendering, while Java takes care of the more object oriented handling. The result is a fairly easy-to-use API that any Java programmer will feel comfortable in.

2.3.2 Structure

When programming with Java3D, one creates a tree structure, a scenegraph, which is built up of a virtual universe with different groups or nodes. The groups contain information of its leaf nodes/groups which make up the scene.

There are different types of groups like BranchGroup, Switch, Locale, TransformationGroup etc. These groups are called Management nodes and manage a collection of child objects.

Further we have the Geometry nodes like Shape3D, Background etc. which defines the geometry in the applications virtual universe.

Also there are Control or Influence nodes which are Behaviours, Sound, Light etc. They define the geometry behaviour.
3 Vizz3D – the Application and its Architecture

Vizz3D is a tool with a complex code structure. It will not be explained in smallest detail here and that is not the intention. This chapter focuses on the basic structure of Vizz3D and some of the functionality. The structure section addresses goal 1 and 2 described in the first chapter. It contains some class diagrams for better understanding. For further explanation of the classes and their calls there are some sequence diagrams in Appendix I.

3.1 The Application

Vizz3D lets the user manipulate and interact with the visualized graph in several ways. One can load graphs add and delete nodes or create a whole new graph. By using metaphors and layouts the visualization of the nodes and edges can be altered. This is explained further in sections 3.2.4-6. One can decide which type of nodes/edges should be visible or not. Picking, rotate and translate (move), zooming etc are some of the features in the canvas.

3.1.1 The GUI

![VizzJ3d](image)

*Figure 3.1 VizzJ3d, the Vizz3D Java3D version. Main GUI - different Panels and the Canvas.*
The GUI (Graphical User Interface) is built up with a mainframe that contains a menu bar at the top, functionality buttons/icons just below, info panels to the left, and the visualization of graphs is put on a canvas (fig 3.1 and 3.3).

The menu bar contains some sub-menus with the same choices as the icons but also the different layouts and metaphors are set from here. Below the menu bar are the functionality buttons (fig 3.2) which let the user use the different features. Icons in the lower row from left to right; New Graph, Load Graph, Save Graph, Save Graph As, (separator), Node/Edge Mode, Add, Remove, Select ,Aggregate, Expand, (separator), Select All, Deselect All, Show all Elements, Hide Selection, (separator), Coordinates, Show Label, Visibility, Show Edge Arrow, Center Node and Rotation.

Figure 3.2 The functionality buttons.

Figure 3.3 VizzGL, the Vizz3D OpenGL version.
The info panels to the left lets the user pick nodes from the lists and displays data information of the picked node.

### 3.1.2 Functionality

The scene (the canvas) can be rotated zoomed and moved around and one can pick objects both with the mouse and from the list panels. Mouse movement over a node gives feedback information of it. A coordinate system, local or global, can be displayed to keep track of the scene transformations. A local coordinate system is located at the graph centre, which can be altered. A global coordinate system is located at the scenes centre.

Aggregation of a node is another feature that collapses all chosen node’s leafs (the sub-nodes) and their edges. The idea is to be able to control the size of the graph and for the user to simplify the graph for better understanding or a clearer view. By choosing the node again and clicking the expand button the leafs become visible again.

An edge arrow is an arrow on the edges midpoint that shows the direction. This clarifies the relation the edges symbolize.

### 3.2 The Structure

The first of the goals was to use as much of the previous code as possible and keep the structures that were already done (since a lot of research has been done in this area during the VizzAnalyzer project). This has been accomplished mainly with abstraction and interfaces. An abstract class is a class that defines and implements some methods. These can be shared by all subclasses that extend the abstract class. The abstraction makes it possible to share the common code, for instance both versions of a node coordinate getter can be identical. An Interface declares some methods that must be implemented by the classes that use it. The interfaces make it possible to call the different versions with the same call. So when calling a Java3D graph one calls it with the same line of code as one would in the OpenGL code by calling a VisualGraphInterface.

When describing Vizz3D’s structure there are numerous ways of doing this. The one chosen here is to roughly divide it into smaller parts:

- **VizzJ3d** and **VizzGL** classes – the application’s starting point where all is run from.
- The **MainFrame** – here is the GUI set up.
- The **Graph** – describing the way the graph is glued together with nodes and edges.
- **Bindings** – decides what type of Metaphors to be used and their appearances.
- **Metaphors** – handles the way a node or edge is embodied.
- **Layouts** – these are algorithms that place the nodes in different ways.
- **Interaction** – this is where all the action takes part, where the graph is painted on the scene.

All these are described in the following sub-chapters. Some sequence diagrams are included in Appendix I for deeper understanding and easier following of the calls.

#### 3.2.1. The **VizzJ3d** and **VizzGL** Classes

These are the applications main classes and from where all is built up. They are derived from the **PlugIn** class, which is the class that every plug-in to VizzAnalyzer must adapt to. They initiate the **MainFrame** and create a **FileLoader** which loads a graph. Then a **GraphicsHandler** is created. The **GraphicsHandler** creates an instance of **MLSceneGraph** (Metaphor and Layout SceneGraph) which, in turn builds up a graph structure. Figure 3.4 describes this with a class diagram. When the graph is created the **GraphicsHandler** adds it to a canvas.
Figure 3.4 The old Vizz3D core structure.

Figure 3.5 shows the new structure with all its interfaces and abstracts. The new class `GraphicsHandlerAbstract` has been added to share common code and `GraphicsHandlerInterface` forces the respective `GraphicsHandlers` to implement some methods.

Figure 3.5 The new Vizz3D core structure. Notice the added interfaces and abstract classes.

The point here is that an instance of `GLGraphicsHandler` can be referred to as a `GraphicsHandlerInterface` and in the same way can a J3dMLSceneGraph be referred to as a `MLSceneGraphInterface`.
3.2.2 The MainFrame

Here is the heart of the GUI. This is where the handling of the user’s interactions is. It checks for interactions with the buttons and lists etc. with the help of an ActionHandler class. At start it sets up an environment in which the application can be visualized in.

3.2.3 The Graph

This is an important part of the application and this section addresses the second criteria of the thesis, the assembling of a graph structure.

First of all, there are three types of graphs mentioned in this thesis and these need some explanation.

1. **The datagraph.** The loading of a graph creates a *datagraph*, the grail. This is a graph that holds metadata, like properties of the graphs nodes and edges. It also is used for iterations and animation amongst other things.

2. **The scenegraph.** This is a legacy from Java3D because of the way Java3D builds up a scene (see section 2.3). In VizzGL it is there to control the making of the visual graph and its metaphors and layouts.

3. **The visual graph.** Throughout the thesis just called the *graph*. This is the structure built up to hold all data needed to be able to visualize and interact with the nodes and edges.

   The respective *MLSceneGraph* creates a graph structure through the *VisualGraphInterface* as a *GLGraph* (or *J3dGraph*). It is a traditional graph with nodes and edges. The nodes visualize the classes in the analyzed code and the edges represent their relations. The graph can have any number of nodes and edges and the edges have two nodes, source and target, attached to them.

   The two respective structures (figure 3.6 and 3.7) show the additional complexity that comes with interfaces and abstract classes. The gain is the object oriented reuse of code and version independent calls through interfaces.

![Figure 3.6](image.png)

*Figure 3.6* The old graph structure before any OpenGL version existed. The J3dGraph extended the J3dGraphskeleton and used HashMap containers for holding the nodes and edges.

When the work on this thesis started the graph structure had been changed to something like figure 3.7 with the exception that the abstract classes did not exist. Abstract classes were introduced to use the methods shared between *VizzJ3d* and *VizzGL*. For instance, in the *VisualGraphAbstract*, getters and setters (methods for setting variables and retrieving their
values) are identical and adding and removing nodes use the same algorithms. The interfaces are there to let some remote method in a class, for instance the Layouts, use a VisualNode Interface as an argument in a getNode call. This way the VisualNode Interface can be both a GLNode and a J3dNode since they both implement the VisualNode Interface.

Figure 3.7 describes the present graph structure with the VisualGraph Interface as an abstract extended the by GL- and J3dGraph Interfaces and also implemented by the VisualGraph Abstract. The GLGraph implements the GLGraph Interface and extends the VisualGraph Abstract. This means that the GLGraph must implement all methods declared in the GLGraph Interface and the VisualGraph Abstract must implement all methods declared in the VisualGraph Interface.

The same approach is used by the node and edge structures. The VisualGraph Abstract has some HashMap containers to keep the graph’s nodes and edges. The compositions in the diagram symbolizes that the VisualNode/Edge Abstract has an instance of the graph and that the VisualEdge Abstract has instances of its source and target nodes.

Figure 3.7 The new graph structure.

The OpenGL and Java3D structures are more or less identical with the exception that Java3D uses a J3dGraph Sceleton class. Further explanation of the graph can be found in chapter 4.
3.2.4 Bindings
The following two sections are the solution to the third goal of the thesis – adaptation to the bindings and metaphors.

The bindings are a way of presenting the graph in different ways. For instance one may decide that all class nodes should be represented by a box or all method nodes should be cones and so on. As a matter of fact this is the core idea of Vizz3D, to be able to visualize code in more familiar ways than just 2D graphics and charts. One can represent nodes with different buildings and edges with roads or just about anything that is intuitive and helps understanding of the code.

The different possible bindings are specified in XML files that are loaded at the application start. At creation of a graph, in the MLSceneGraph, a default binding is created where the different node types are bound to a specific metaphor, for instance a sphere or a cone.

3.2.5 Metaphors
When visualizing nodes and edges one have to decide what kind of shape and appearance they should have. This is solved by using Metaphors. They can be a Sphere, Cone, Cylinder, Box, Line or a Mesh. A mesh shape is a shape built up with polygons in some other 3D construction application (for example 3DS Max) as long as it comes in 3DS format.

Java3D have predefined shapes with Appearance and Material. The information for these are gathered from the loaded graph and stored in the shape. For VizzGL one had to create a similar solution where every node and edge has an instance of MetaphorTypeInterface (fig 3.8). It is a GLMetaphorType which derive into any of the GLShapes (e.g. GLBox, GLLine etc.). The GLMetaphorType has an instance of GLAppearance that holds the shape data (i.e. width, length, radius etc) and also a GLMaterial which is a holder for color.

![Diagram of Metaphor Types](image)

Figure 3.8 The metaphor types.

3.2.6 Layouts
As explained before the layouts are algorithms that decide the nodes position in the graph. It is an important feature of the application since it helps structure the graph in a logical way. This thesis
will not go deep into the Layout structure since this section was out of scope but all the layouts are derived from the LayoutEngine and they implement the LayoutInterface. Through the GUI the layout is decided and then linked to the graph and its nodes.

### 3.2.7 Interaction

The following sub-chapter briefly explains the structure of the interaction part of Vizz3D. This is all new code and the interface and abstraction is there for a few shared methods and version independent calls (fig 3.9).

The Interaction is the part where all picking, transformation, rotation and all other interactivity is handled. It is also where the actual drawing on the scene is carried out, therefore it is a section filled with algorithms and complex calculations demanding on the CPU. This is actually where all the OpenGL code is and where a lot of this thesis’s work is based at.

The GLInteraction is created by the MLSceneGraph. Then a GLAdvancedInteraction is created and this is where all painting and interaction on the canvas is controlled. The abstract class GLAnimCanvas is a predefined GL4Java class which is used to set up a canvas, which will be explained more thoroughly in the fourth chapter.

![Interaction Diagram](image)

**Figure 3.9** The GLInteraction constructor creates a GLAdvancedInteraction which is called to through the InteractionImplInterface

GLDrawingFunctions is a class where all the shapes are drawn. Also the Edge arrows are drawn here. GLScenePropInterface is an interface that declares some static properties like the materials, lights and lines patterns.
4 The Implementation

This chapter will cover the actual work that is done, the implementation and addresses more or less all of the thesis’s sub-goals mentioned in chapter 1. A lot of time has been spent refactoring the existing code into a version that works with OpenGL and some code, in the interaction section, are completely new. Figure 4.1 shows a run of the new version.

![Image](image.png)

**Figure 4.1** *Vizz3D*, the OpenGL version.

4.1 Setting up the canvas

The first thing one has to do when creating a GL4Java scene is setting up the environment. The place in this case is the GLAdvancedInteraction. Since this is an animated scene (i.e. it is not a static picture) the GLAnimCanvas class is used as a base class. There are a couple of methods, described in the table below, that is called automatically when the GLAnimCanvas is used, and can be overridden to fit the needs.
### preInit()

This is a method that is called before the GL Context is created. Here one sets variables like double buffering and frames per second (fps). In this implementation the different fps values are set depending on the graph's size. The bigger the graph the fewer the fps. This has to do with the fact that a large graph is very demanding on the CPU and a restriction had to be done. A higher fps results in a smoother animation, but is costly.

### init()

This method is used to set state variables like light, fog, fonts, anti-aliasing, depth testing, background color and so on. Three point lighting system is used; one main light placed in the upper right corner in front of the graph, one warm and one cold light that are placed behind the graph to give the scene some depth and a more realistic feel. The lights will be returned to later when the rotations and translations are covered.

The init method is also the place to create display lists which are a way to predefine shapes in order to optimize the calculations made every frame. There have been no time to investigate this further but since there only are simple base shapes used in this application (so far) this would not have generated any large wins in terms of time. But it will surely become useful once the mesh shape is used.

Cull facing is set here too. This means that only the exterior of shapes are rendered, which is really saving a lot of CPU power. In this application transparency is not used that much but a box would lose depth if the interior wasn’t rendered, so this has to be thought of in case of transparency.

The last thing in this method is a start() call which tells GLAnimCanvas that initialization is done and everything is set in motion.

### display()

Here all rendering is done. This is split up in two parts; renderPickableObjects() and renderUnpickableObjects(). The reason for this is that during picking operations one renders the scene (in selection mode) to see which objects are picked. This way one can restrict this to the relevant objects. After the rendering is done the picking() method is called. This will be discussed in more detail later. Some other calls typical for OpenGL code is also made like clearing and swapping the buffer, flushing and so on.

### reshape()

This method sets up the volume which is the scene space. It specifies the field of view, which is the angle the camera or eye can see and the volume boundaries used.

### doCleanup()

This is where all the cleaning up is done and it is called when the canvas is closed.
4.2 Picking
When talking about picking in this application one have to separate pickings done passively by moving the mouse over objects and active picks by clicking an object. The first is used for mouseOver effects like showing a nodes label. The latter is for setting an object as marked for later actions (e.g. centering, aggregation etc).

The picking is basically carried out by defining the viewport, what is actually seen, and then defining the picking window, inside which area should one pick and then deciding which of the picked objects one should consider “picked”. There are two ways of picking in this application; by clicking an object and by dragging a pick box. In the first case a small rectangle (4x4 pixels) is checked. If any objects are inside they are put in a buffer. If more than one object is picked the one closest to the viewer is considered picked. When using a pick box all objects inside the picking area is considered picked.

4.3 Transformations
This section includes the solution of the fifth goal of the thesis, the interaction.

Rotations, translations and scaling is what makes the scene get a real interactive 3D feel. They make it possible to rotate and move around in the scene and are therefore an essential part of the OpenGL code. And, as we shall see, can be quite tricky. OpenGL have blessed the users with a handful of predefined operations that really helps, otherwise all matrix calculations would have been done “by hand”. Nevertheless one has to know what happens to control the transformations.

The three basic operations are carried out in a special order; translations first, rotations second and scaling last. Any other order ends up with wrong results. For instance say you scale first by 0.5 over all axis’s and then translate and rotate, then the translation will be affected by the scaling and only moved half the way it should.

The transformations are placed before the actual drawing is carried out.

4.3.1. Translation
Translations are used to place the nodes, edges and other objects in position but also are used to move the whole graph.

One problem that occurred with the movement of the graph was that it did not move intuitively. Say, for instance that the scene had been rotated and the z-axis points to the left on the screen. When dragging the mouse, all one gets out of it is x and y movement and therefore the movement would result in the graph closing up on the viewer or it would distance itself. A way to solve this is to check with the scene what a mouse movement would represent in the scene. The way to do this is by using the gluUnProject method (fig 4.2). This method maps specified screen coordinates to world coordinates.

The gluUnProject method is used twice, one for the present mouse position and one for the previous position. The difference gives us the answer to where to translate the graph, its new coordinates.
Point tempPoint = e.getPoint();

if (mousePrevPoint == null) {
    mousePrevPoint = tempPoint;
    return;
}

double[] modelMatrix = new double[16];
double[] projMatrix = new double[16];
int[] viewport = new int[4];

// set up the current view to work with
gl.glPushMatrix();
gl.glLoadIdentity();
gl.glTranslatef(xTrans, yTrans, zTrans);
doRotation();

// Get the current matrices and the viewport
gl.glGetDoublev(GL_MODELVIEW_MATRIX, modelMatrix);
gl.glGetDoublev(GL_PROJECTION_MATRIX, projMatrix);
gl.glGetIntegerv(GL_VIEWPORT, viewport);

double[] world = new double[4];
Point3f worldTemp = new Point3f();
Point3f worldPrev = new Point3f();

// Transform the current mouse position from screen to world
glu.gluUnProject(tempPoint.x, viewport[3] - tempPoint.y, 0,
modelMatrix, projMatrix, viewport, world);
worldTemp.x = (float) world[0];
worldTemp.y = (float) world[1];
worldTemp.z = (float) world[2];

// Transform the previous mouse position from screen to world
glu.gluUnProject(mousePrevPoint.x, viewport[3] - mousePrevPoint.y,
0, modelMatrix, projMatrix, viewport, world);
worldPrev.x = (float) world[0];
worldPrev.y = (float) world[1];
worldPrev.z = (float) world[2];

double diffX = worldPrev.x - worldTemp.x;
double diffY = worldPrev.y - worldTemp.y;
double diffZ = worldPrev.z - worldTemp.z;

Figure 4.2 A code snippet that shows the usage of the gluUnProject method

4.3.2. Rotation

Rotation in the scene is done by dragging the mouse and holding the left button down. Every
frame the mouse position is checked. If it is moved in any axis (x or z) the xRot and yRot is
affected (Fig 4.3). These are variables that are set to 0 at start and then used when the rotation
operations are done. Note that this affects the whole graph, not just a single object.
Point tempPoint = e.getPoint();

if (mousePrevPoint == null) {
    mousePrevPoint = tempPoint;
}

// the 0.7 is just a factor to make rotation slower/smooth
if (tempPoint.x > mousePrevPoint.x) {
    yRot += 0.7 * Math.PI;
} else if (tempPoint.x < mousePrevPoint.x) {
    yRot -= 0.7 * Math.PI;
}

if (tempPoint.y > mousePrevPoint.y) {
    xRot += 0.7 * Math.PI;
} else if (tempPoint.y < mousePrevPoint.y) {
    xRot -= 0.7 * Math.PI;
}

mousePrevPoint = tempPoint;

Figure 4.3 Code that adjust the xRot/yRot according to the mouse position.

This way of rotating the scene does not result in the same acting as in the translation since one
work with only two axes. The solution could be to use the same solving method as in the
translation. The rotation nevertheless works as it should, like VizzJ3d.

4.3.3. Scaling
There is only one place where scaling is used in this application and that is in the renderNodeText
method. It is used to size the text. It is a solution where the graphs distance to the viewer is used
to decide how much the text is to be scaled.

A nicer solution might be to use the gluProject method that is the opposite of the
 gluUnProject described in section 4.3.1. This would result in a text field always the same size
and always in front of the graphs different objects.

4.4 The Graph
The graph class is the base in which the nodes and edges are located. This is done in two ways.
Firstly, in the GLGraph class the GNodes andGLEdges are saved in vectors, a nodeList and an
edgeList. The vector class is used since it is indexed with integers and, at picking; an integer is
what OpenGL gives as a result. Secondly, two HashMaps each are used to store the nodes and
edges respectively. This ensures fast access when calling with different arguments. The second
storing is used by the system in general while the first is used by the GLAdvancedInteraction
class.

4.4.1 Nodes
The node structure holds a lot of information; its coordinates, its visibility, is it collapsed or not,
is it marked, its metaphor and appearance. A node can be aggregated, that is when all children to
the node is collapsed into it, or in practice, made invisible.

In GLAdvancedInteraction the nodes are reached through graph, a GLGraphInterface. In the
renderNode method the node is checked if it is visible or not. If not, no further action is done.
Then the lights are set. The reason for doing this here is that the light should have the same
position on every node, otherwise it results in strange effects when rotating the scene and viewing
from different directions. After this the nodes materials are set, fetched from its appearance. If it
is marked the emissive color is set to give it a “glow”. Then the node is translated and rotated into its position in the graph. The next step is the actual drawing. The shape is depending on what type of metaphor the node has. All shapes are drawn in the \texttt{GLDrawingFunctions} class.

### 4.4.2 Edges

Edges are of course the glue between the nodes, describing their relations. Their structure also holds some information; metaphor and appearance, source and target, visibility, marking and locking (both for future implementations) and if the direction arrow is to be shown or not.

The \texttt{renderEdge} method checks the visibility just like the nodes and then gets the source and target coordinates. Then the method sets the lights and colors and checks for the edges pattern. If a direction arrow is to be drawn it is done just before the actual drawing which is the last thing to happen in the method.

### 4.4.3 Edge Patterns

The edges have different patterns to symbolize the type of relation their nodes have to each other. This is done with a 16 bit short number. Each pattern has a short number that represents the decimal translation from binaries. For instance if we want a dashed pattern it is drawn with 8 filled pixels followed by 8 empty ones. Also, to confuse it a little bit, OpenGL calculates the short number backwards for optimization reasons (see [4] p. 96), so the binaries are flipped or “mirrored”. Then we have a binary string as the table below.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Binary</th>
<th>Short number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dashed</td>
<td>0 0 0 0 0 0 0 1 1 1 1 1 1 1 1</td>
<td>255</td>
</tr>
<tr>
<td>Dash-dotted</td>
<td>0 0 0 0 0 0 0 1 1 1 1 1 0 1 1 1</td>
<td>2175</td>
</tr>
<tr>
<td>Dotted</td>
<td>0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1</td>
<td>257</td>
</tr>
</tbody>
</table>

If one would translate this binary the result is 255 which then is used when setting the \texttt{glLineStipple} method. The remaining patterns are calculated in a similar way depicted in the table. A solid pattern is treated as a normal line and line stippling is disabled.

### 4.5 Edge arrows

To simplify the understanding of the edges in the graph, there should be some sort of explanation of their direction to clarify the relation the two nodes have between them. The wanted result was to have an arrow pointing out the direction in the edges midpoint. This was quite a tricky problem that took some mathematical research [8] and in the end this was the solution to the fourth goal of the thesis. See figure 4.4 and 4.5 describing the solution.

To begin with there are a couple of known facts and variables that one could use. The “default” drawing direction (i.e. if one draws a cone “on the spot”) is pointing in the z-axis direction, which initially is towards the viewer and so the first vector, called \texttt{zAxis} is (0,0,1). Then there is the edge vector (\texttt{edgeVector}) which is (dx, dy, dz) or the difference between two points on the edge (here the edge’s source and target coordinates are used but it does not really matter as long as one gets the right direction). The edge vector is normalized to make the matrix calculations easier for the hardware. Normalization means that the vector is adjusted to the unity circle.

Now, from these two vectors one can get the angle between them. It is called \texttt{angle}. Then the axis to rotate about needs to be found and it is the cross product of the two known vectors. This is called the \texttt{rotVector} and is perpendicular to the plane that the \texttt{zAxis} and \texttt{edgeVector} build.
Regarding the code snippet below, the angle is calculated with the help from the vector package `javax.vecmath.Vector3d`. Otherwise the `edgeVector` would have to be translated into origin. The midpoint is calculated for translation into drawing position. The actual arrow is drawn using a predefined wire cone method.

```java
double midPointX = (from[0] + to[0]) / 2;
double midPointY = (from[1] + to[1]) / 2;
double midPointZ = (from[2] + to[2]) / 2;

Vector3d zAxis = new Vector3d(0.0, 0.0, 1.0);
double dx = to[0]-from[0];
double dy = to[1]-from[1];
double dz = to[2]-from[2];

Vector3d edgeVector = new Vector3d(dx, dy, dz);
edgeVector.normalize();
double angle = Math.toDegrees(zAxis.angle(edgeVector));
Vector3d rotVector = new Vector3d();
rotVector.cross(zAxis, edgeVector);

gl.glPushMatrix();
gl.glTranslated(midPointX, midPointY, midPointZ); // move to position
gl.glRotated(angle, rotVector.x, rotVector.y, rotVector.z); // do rotation
glut.glutWireCone(r, 3*r, 3, 1); //base, height, slices, stacks
gl.glPopMatrix();
```

Figure 4.5 Calculating the angle and axis to rotate about.

### 4.6 Feedback Text

In order to understand the graph at visualization identification of the nodes are a necessity. The solution is to have some kind of text feedback when hovering over the node with the mouse, a mouseOver effect. It is accomplished by, whenever the mouse moves over the canvas, activating the picking. If the mouse is over a node, a boolean variable (`isShowName`) in the respective GLNode is set to true.
In the `renderUnpickableObjects` method all nodes are checked and any eventual text is drawn in the `renderText` method.

The `renderText` method checks for node visibility first and returns if the node is invisible which can happen if the node is aggregated or manually put in invisible mode. If visible we continue with translation to the nodes coordinates. Then a reverse rotation is done to make the text face the viewer. By checking the metaphor a translation to the shapes outer limits is done, for instance if it is a sphere one can use the radius.

The actual drawing of the node text is done with the GLF API [9] which is an OpenGL library for rendering fonts in various ways.

4.7 Metaphors, Appearance and Materials

Where Java3D have predefined shapes with appearances and materials, OpenGL leaves this for the programmer to solve the way it suites him. The simplest way to set materials in OpenGL is to define the color just before drawing a line, polygon or other shape. The VizzGL implementation uses the `GLAppearance` and `GLMaterial` classes for color storage in the `GLNode` or `GLEdge` and is called through them respectively.

The different material settings shown in the code snippet below (fig 4.6) is diffuse, ambient, specular and shininess which respectively represent their base color, darker areas, lighter areas and how shiny the should appear, their glossiness. Also if the node is marked an emissive color is set. That results in a glowing appearance.

As described in section 3.2.4 the bindings control the metaphors appearance. The appearance is unique for each object in the scene depending on its characteristics. For instance a large class node can be a red sphere with a large radius whereas a smaller method node is displayed as a tiny blue box. Of course the metaphor rules whether it is the attribute radius, height, width etc that is set and used. The creation of a metaphor is done in the `GLMetaphorFactory`.

```java
GLAppearance app = new GLAppearance();
app = graph.getNode(i).getApp();

GLMaterial nodeMtrl = new GLMaterial();
nodeMtrl = app.getMaterial();

if (graph.getNode(i).isMarked()) {
    gl.glMaterialfv(GL_FRONT, GL_EMISSION, nodeMtrl.getEmissiveColorArray());
}

if (glMetaphor instanceof GLBox) {
    drawFuncs.drawBox(app.getHeight(), app.getWidth(), app.getDepth());
} else if (glMetaphor instanceof GLCone) {
    drawFuncs.drawCone(app.getRadius(), app getHeight());
} else if (glMetaphor instanceof GLCylinder) {
    drawFuncs.drawCylinder(app.getRadius(), app.getHeight());
}
```
else if (glMetaphor instanceof GLSphere) {
    drawFuncs.drawSphere(app.getRadius());
}

Figure 4.6 A node’s material settings followed by a metaphor check and drawing. Notice the different appearance as arguments depending on the metaphor.
5 Conclusions and Future work

This last chapter will summarize the thesis and follow up the solving of the goals and criteria. It will also take a look at what could be bettered and future work in the VizzAnalyzer area.

5.1 Conclusions

To approach a framework with the size of Vizz3D is quite overwhelming at first. A large amount of time has been spent on understanding the code, both from the Java perspective and the Java3D point of view. Especially Java3D gave me some trouble since it has quite a different thinking from OpenGL.

A problem during the work with this project has been the constant changes of structure and in some cases functionality. Vizz3D is under construction by more than one person and this leads to difficulties and delays. The problems were overcome with the help from the supervisor and a large amount of patience.

To integrate OpenGL where Java3D had been the model proved quite difficult, but possible. The main difficulties were the usage of all interfaces and abstracts and to merge new code with the existing framework. The solutions often were, as mentioned before, the creation of interfaces and abstract classes to with sub-classing. This resulted in a proper object oriented code with the possibility to use these classes in future versions.

The functionality is more or less identical to VizzJ3d. Some differences in the appearance occur but that is considered of less importance. For instance the VizzJ3d marking of a node results in a pulsating wire sphere around it whereas in VizzGL it highlights.

Finally: All the problems of this thesis have been tackled in the previous chapters and sections. The solutions to the sub-goals stated in section 1.4 are summarized in the following sections.

5.1.1 Refactoring and Integration

The solving of this goal is discussed in chapter 3. The application architecture has changed dramatically under the process of this thesis. The result is a code structure that allows new versions of Vizz3D to be implemented with much less effort. This is of great importance since “wheel invention” can be avoided this way. All methods used by both versions are placed in abstract classes. Some classes have been copied and placed in the VizzGL related packages and then adjusted to VizzGL.

The OpenGL code in the interaction structure has been split up into a few classes and interfaces. This concept could be developed further but the time made this impossible. The main idea of how it is done is solved though and could be used in future implementations.

5.1.2 Assembling

This solution is presented in section 3.2.3 and 4.4 which describe the structure and its implementation respectively. The graph structures used in Vizz3D did undertake some extensive refurbishing but in the end VizzGL does not use a scenegraph the way Java3D does, and it should not. Java3D uses the scenegraph for convenience reasons; OpenGL does not for optimization reasons.

The developed structure is using abstracts and interfaces discussed earlier. It is now adapted to fit the needs of both versions.
5.1.3 Adaptation
This is discussed in sections 3.2.4-5 and 4.7. By creating a similar structure to the one Java3D uses predefined Shapes, with their appearances and materials, the integration of bindings were made easier. With the creation of the GLMetaphor structure, with the GLMetaphorFactory as the creator of a metaphor, and the GLAppearance and GLMaterials calls to these classes can be done in the same way by the system. All with the help from the GLBindings structure which is refactored and adjusted to VizzGL.

Two things were not implemented in VizzGL; mesh shapes and textures. Both of them are important to implement in the future since it is one of the main ideas to visualize with familiar contexts (see the introduction). The reasons for not implementing it now were lack of time and the fact that it is not used in VizzJ3d at the moment.

The usage of metaphors and bindings is nevertheless working which was the third of the sub-goals.

5.1.4 Edge arrows
The solution is presented in section 4.5. A direction arrow is drawn in the edges midpoints. This was a problem solved with some vector and angle calculations. An implementation with 3D graphics should, one feels, generate a lot of mathematical formulas and calculations. But the fact is that this is more or less the only place, take away the layout algorithms, in VizzGL implementation where any advanced math is visible. OpenGL lets the implementer use predefined methods for transformations and their matrix calculations.

The edge arrows are now visible if the user so chooses. This is good since a graph with hundreds of edges and their arrows can be difficult to read because of the large quantity of objects. But the edge arrows are a necessity for explanation of the nodes relationships.

5.1.5 Interaction
The use of the OpenGL method gluUnProject was the solution presented in section 4.3. It helps finding the coordinate representation of the mouse position in the scene. This was implemented for the translations and should, in the future, also be done with the rotations. This would generate a natural rotation, around all axes’, that will react to the user’s interaction independent from earlier rotations and their rotation angles. Nevertheless the result is pleasing and the goal is achieved.

5.2 Future Work
The present implementations in OpenGL (GL4Java) and Java3D works well but, as any implementation, it can be improved. More interaction functionality would increase the user’s possibility to understand the code structure. Transformations of single nodes are already prepared but not fully implemented, enhanced rotation around three axes, the use of displaylists in VizzGL is a few improvements that could be considered. Future implementers may use the sequence diagrams in Appendix I as a help.

Other implementations are investigated at present. JOGL [10] is an open source GL4Java-like API developed at Sun Microsystems [12]. JOGL gives the Java programmer full access to the native OpenGL 1.5 API and nearly all vendor extensions (GLX, GLF etc.). Another API considered is the Xith3D [11] which features are almost identical to Java3D and its scenegraph structure. The difference is that Xith3D allows native calls to OpenGL with the help of JOGL.
References

Appendix I – Sequence diagrams

This section displays some sequence diagrams for deeper understanding of the code structure and the calls between the classes and their methods.

![Sequence Diagram](image)

**Figure A1.1** A simplified sequence diagram showing what happens in the *VizzGL* class.
Figure A1.2 The GraphicsHandler and its actions.
Figure A1.3 Setting up the GUI.
Figure A1.4 The creation of the graph structure.
Figure AI.5 The BindingManager.

Figure AI.6 The default binding is set at start.
Figure A1.7 Adding shape to a node
Appendix II – Screenshots

This second appendix shows some additional screenshots, taken from both versions presenting them in different situations.

Figure AII.1 Java3D version. Three nodes are marked and one displays the feedback text with label and node type.
Figure A1.2 Different bindings results in different appearance. Compare it with figure 3.2.